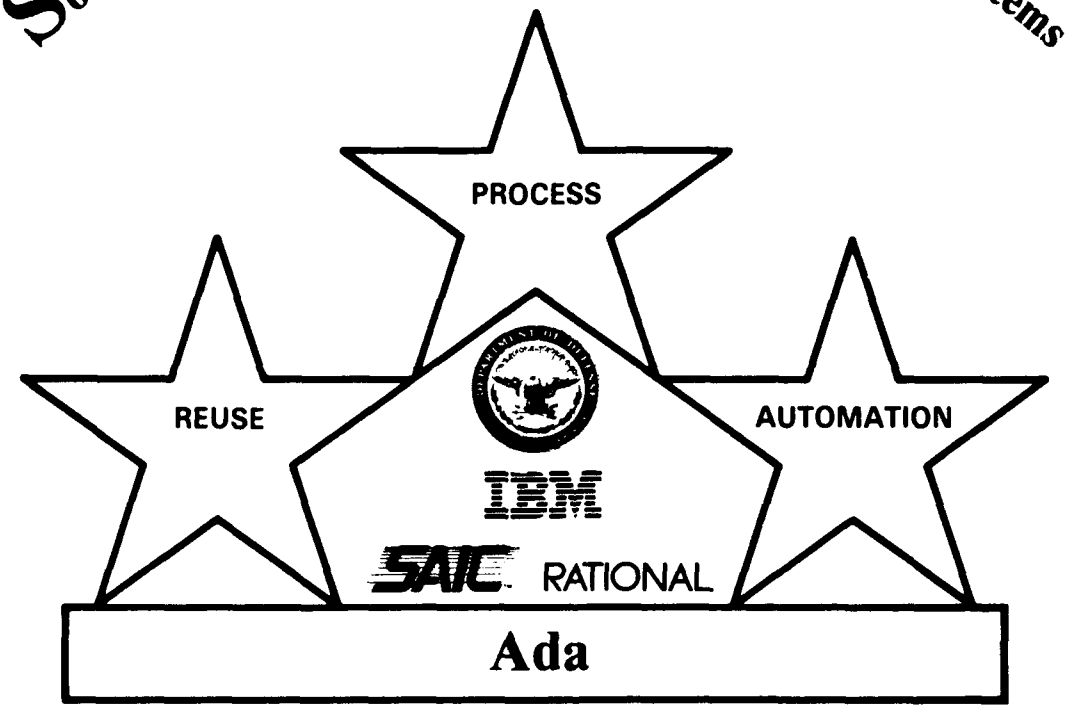
Document No. 2000-001
31 December 1989

General Definition of Project (Ada / SQL Binding)

for the

Software Technology for Adaptable Reliable Systems



Contract No. F19628-88-D-0032

Task IR67 – SQL / Ada Program Language Interface

CDRL Sequence No. 2000

31 December 1989

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 31, 1989	Final

4. TITLE AND SUBTITLE
General Definition of Project (Ada/SQL Binding)

5. FUNDING NUMBERS
C: F19628-88-D-0032

6. AUTHOR(S)
S. Phillips

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IBM Federal Sector Division
800 N. Frederick Avenue
Gaithersburg, MD 20879

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
CDRL Sequence No. 2000

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A standard binding was needed between Ada and SQL (Structured Query Language), the ANSI and DoD standard for accessing commercial relational data base management systems (DBMS's). SQL was not designed to be embedded within applications in general purpose programming languages, such as Ada. Previously developed Ada-SQL bindings have had various technical drawbacks.

A prototype Ada-SQL binding was built by automating the SQL Ada Module Extension methodology (SAME). SAME is a method for building Ada applications that access DBMS's via SQL. SAME extends SQL by exploiting the features of Ada.

This Technical Plan presents the background, technical approach, and top-level capabilities of the project. It also discusses such technical problems as storing arbitrary data types in a data base and using SAME without a module language compiler.

Kaywood; STARS (Software Technology for Adaptable Reliable Systems). (KR)

14. SUBJECT TERMS	15. NUMBER OF PAGES
STARS, Ada, SQL, Structured Query Language, data base management system, DBMS	21
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

General Definition of Project
(Ada / SQL Binding)
for the
Software Technology for Adaptable, Reliable Systems (STARS) Program

Contract No. F19628-88-D-0032

Task IR67 – SQL / Ada Program Language Interface

CDRL Sequence No. 2000

31 December 1989

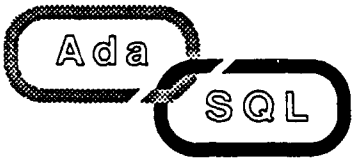
Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

IBM Systems Integration Division
800 North Frederick Avenue
Gaithersburg, MD 20879

Technical Plan
for
Ada/SQL Binding Supporting the SAME Methodology



Contract Number: F19628-88-D-0032/0002
CDRL Number 2000

December 3, 1989

Submitted to:

IBM Corporation
Systems Integration Division
800 N. Frederick Road
Gaithersburg, Maryland 20879

Attn: Ray Grimes

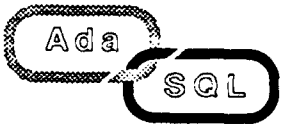
Lockheed
Missiles & Space Company, Inc.
Software Technology Center
2100 East St. Elmo Road
Org. 96-10 / Bldg. 30E
Austin, TX 78744
(512) 448-5740

E-mail: phillips@stc.lockheed.com

A Technical Plan for Ada/SQL Binding Supporting the SAME Methodology

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS



A Technical Plan for Ada/SQL Binding Supporting the SAME Methodology

The Ada/SQL Binding project will implement a binding between the Ada programming language and a relational data base management system, specifically by automating the SQL Ada Module Extensions (SAME) methodology as described in *Guidelines for the Use of the SAME*, an SEI Technical Report [1]. This is the technical plan, CDRL 2000, for implementing a prototype binding supporting that methodology.

BACKGROUND

Conventional approaches for binding ANSI-standard SQL to Ada allow embedding SQL statements directly into Ada programs, thereby creating something that is neither pure SQL nor pure Ada. A preprocessor is used to remove the SQL statements and replace them with valid Ada subprogram calls. However, direct access to the data base is still part of the application program. By using SAME approach, these two contexts, SQL statements and Ada statements, are separate modules thereby implementing a modular approach to data-base definition and access. This modular approach allows the efficiency of having programming tasks assigned to programmers who specialize in each area.

SAME is a method for the construction of Ada applications that access data base management systems whose data manipulation language is SQL. As its name implies, SAME extends the Module Language defined in the ANSI SQL standard *Database Language—SQL* [2] by exploiting the capabilities of Ada. The defining characteristic of the module language is the collocation of SQL statements, physically separated from the Ada application, in an object called the *concrete module*. SAME treats the module much the same as it treats any other foreign language; that is, it imports complete modules, not language fragments.

SAME provides the binding between these two modules through an interface layer, called the *abstract module*. The abstract module serves to transform data from abstract definitions to concrete types (and back again). The abstract module makes calls to an Ada specification representative of the SQL module (the *concrete interface*).

During application design and development, SAME is used as follows:

- The abstract domains that occupy data base columns are defined and described as Ada types. This is done using standard packages available to users of SAME methodology.
- The application programmer, along with the SQL programmer, determines the services that will be needed from the data base. They are coded in SQL and collected in a concrete module .

- An abstract interface is created. This is a set of package specifications declaring the record type definitions needed to describe row records and the procedure declarations needed to access the relevant concrete module procedures. This interface will be called by the Ada application.
- The type within the abstract domain definition must be determined for each data item at the abstract interface.
- The abstract module, the bodies of the procedures declared in the abstract interface, is created.
- The application program is written. It can be written once the abstract interface specification is completed and concurrently with the creation of the abstract module bodies because the application does not need the module bodies to compile against.

PROJECT DESCRIPTION

The purpose of this effort is to produce a system that will allow an Ada program to talk to a commercial data base management system (DBMS). The delivered packages can be tailored to support any SQL-based DBMS. The first prototype will be developed on a Digital VAXstation 2000. This hardware choice was driven by the fact that a production-quality SQL module language compiler and its associated relational data base management system are available on this platform (as is an acceptable Ada compiler and development environment). Currently, this is the only readily available system with all needed software. The module language compiler could be simulated using one of many DBMS systems that support embedded SQL. However, SAME was intended for use with a module language compiler, which makes this choice a valid one. Figure 1 shows the configuration of an Ada application accessing a DBMS using SAME.

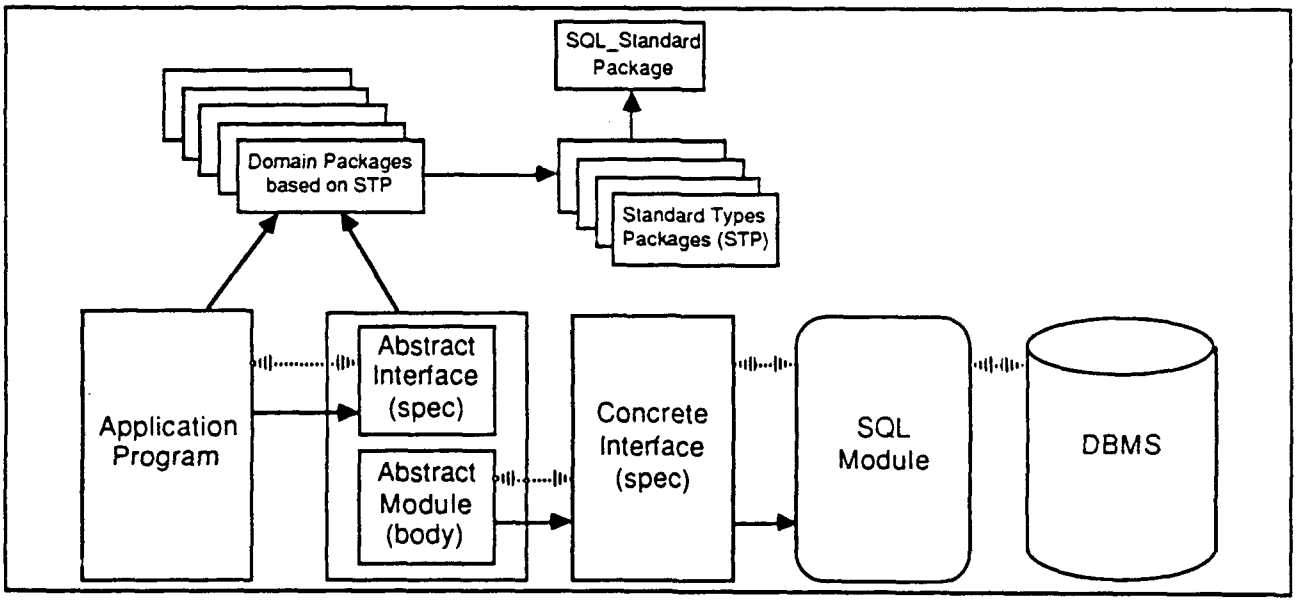


Figure 1. Configuration of SAME Methodology. *(Black arrows indicate Ada visibility; hatched arrows show data flow.)*

TECHNICAL APPROACH AND TOP-LEVEL DESCRIPTION OF PROPOSED FUNCTIONAL CAPABILITIES

The approach for this project is based upon previous work completed for STARS Foundation Contract N60921-87-C-0293. That effort used input defining the data base structure gathered from a data base administrator (in the form of an Ada procedure that instantiated many generics) and automatically generated an Ada package for use by the application programmer and valid SQL statements that provided data definition of the required tables. This project will receive the same form of input from a user (in this case, an Abstract_Interface programmer) and automatically generate domain packages, the abstract interface, and the abstract module. It avoids preprocessors, and all of the activity necessary to support it is hidden from the application programmer. In this scenario, the interface programmer provides the interface that is to be `WITH`ed by an application. In order to provide the application programmers with these facilities, the interface programmer must understand both the data base structure and each application's needs.

Providing automatic generation of several packages from one source offers several advantages:

- The work load is less. By writing and executing only one Ada procedure, many packages are automatically generated.
- All of these packages are related, and generating them from one source guarantees compatibility.
- All necessary support for the automatic generation is hidden from the Ada and SQL programmers. If the underlying data base is changed, the Ada application may not even need recompilation.
- This structure requires no preprocessing of source code in any language, which eliminates another source of error and guarantees that any certified Ada compiler will be able to provide this function. This is achieved through the use of Ada generics.

The first three points are goals of all such efforts and should be considered minimum requirements. The fourth avoids preprocessors and depends heavily on nested Ada generics. Examples of Ada code are provided later in this document to describe this approach..

The interface programmer initiates the process of making the data base available to the Ada programmer. The first step in this process will require the interface programmer to write an Ada procedure, following specific rules and guidelines, that automatically creates the domain packages used by the application. The first step in writing this description is to instantiate the Abstract_Domain_Generator generic with two parameters, one defining the name of the domain

package and the second naming the columns within the data base that are being defined as abstract domains. This generic will be instantiated for each separate domain package.

The second step in this procedure will be to instantiate the Domain_Generator generic for each domain specified in the initial instantiation. This method provides enough redundancy to check for undefined or multiply defined domains (or columns). The generic is instantiated with the domain name, its class, range values, and whether or not the type supports null values.

When this Ada procedure is compiled and executed, syntactically valid Ada packages that support SAME abstract domain semantics are automatically generated. These are compiled and made available to the application programmer. Also, an Ada package specification Base_Specific_Domains is automatically generated to be used by the interface programmer as he completes his task by automatically generating the Abstract_Interface (Ada package specification and body).

To generate the Abstract_Interface, the interface programmer will once again write an Ada procedure that instantiates several generics. This procedure will `WITH` the previously generated Ada package specification Base_Specific_Domains that contains an enumerated type of all valid domains. This will ensure consistency between parameters of the procedures of the abstract interface and the actual domains.

The first step in writing this procedure is to instantiate the Abstract_Interface_Generator generic. It is instantiated with the name of the abstract interface package, an enumerated type representing the row record names to be defined, and an enumerated type representing the procedure names that will make up the interface. Once this is instantiated, subsequent nested generics will be instantiated with objects of these two enumeration types, ensuring semantic consistency. The next level of generic is instantiated for each row record. The generic parameters are the row_name (of the previously defined row-record-names type) and an enumerated type representing the name of each record component. This is followed by an instantiation of generic package Generate_Record_Component for each component in the record.

Still within the Abstract_Interface_Generator, the next step is to instantiate the Generate_Procedure generic. The parameters to the generic are the procedure name and an enumerated type representing parameter names. This is instantiated once for each procedure in the interface. The next level generic is instantiated once for each parameter of the procedure. This generic is instantiated with the parameters name, its type, and its mode. By nesting generics in this manner the implementation is provided with enough semantic knowledge of the structure of the Abstract_Interface so that most incomplete definitions, missing definitions, and semantically invalid definitions are caught at compilation. After successful compilation and execution, an Ada

package representing the Abstract_Interface has been automatically generated. The package is compiled and made available to the application programmer. It is this package that will interface the DBMS and the Ada program. Figure 2 represents the detailed architectural structure of the proposed system. Each functional component is further explained in the next section.

Packages

The following paragraphs describe the packages in Figure 2 in detail. SAME standard packages are not included since they are not being developed as part of this contract.

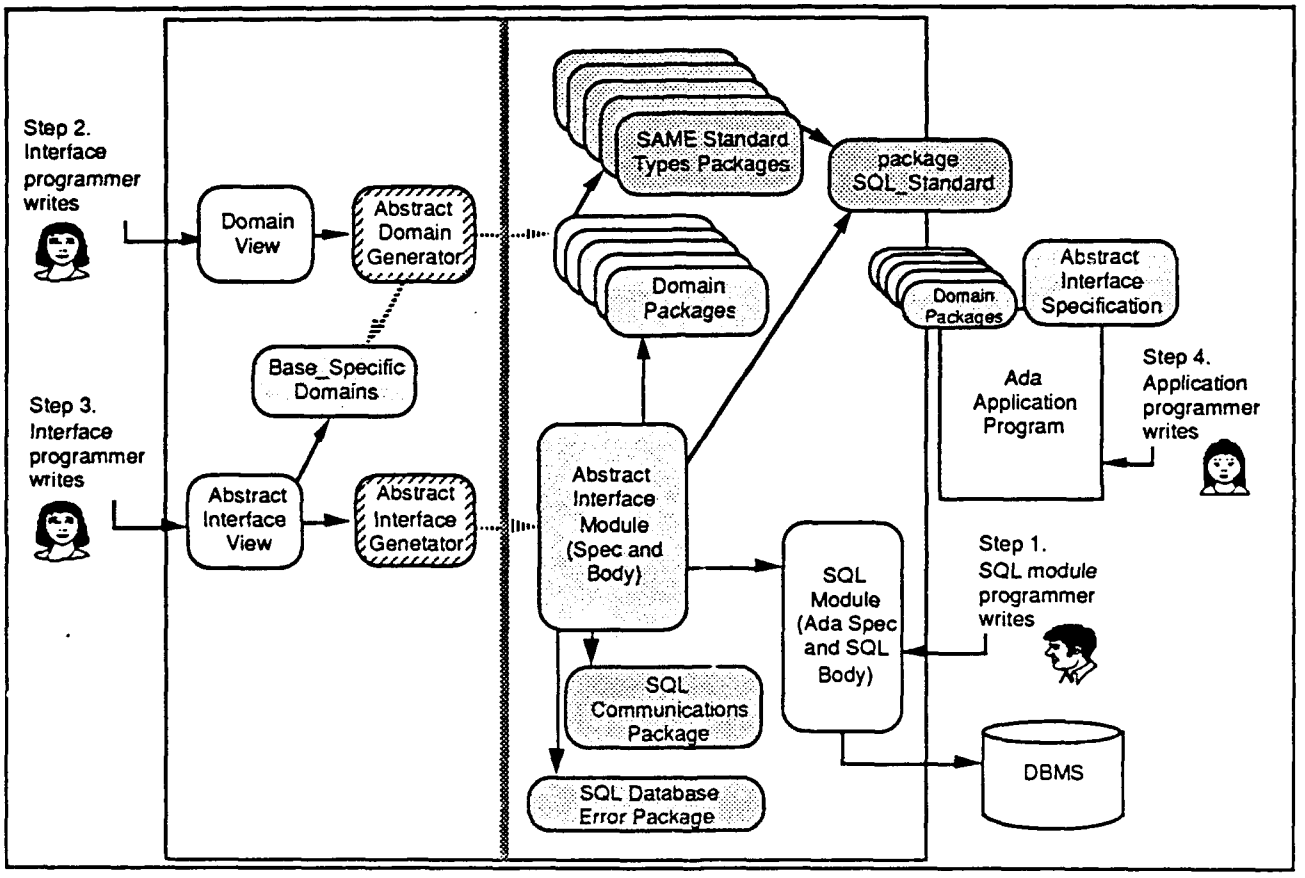


Figure 2. Detailed Architectural Structure of the Proposed System. (Hatched lines indicate generation, with the arrowhead pointing to the generated code; solid lines show Ada visibility with the arrowhead pointing to the WITHed code. Dark ovals indicate SAME packages, and lighter ovals show automatically generated code. Unshaded ovals show manually input, and hatched ones show code to be provided for the binding. Packages used only during development appear to the left of the vertical gray bar; packages used at run time appear to the right.)

Domain_View

Domain_View is an Ada procedure written by the interface programmer to define the domains needed by the application programmer. When this procedure is compiled and executed, Ada packages representing the domains are automatically generated for use by the application program.

As such, it is not a deliverable of this project, rather the rules and methods for writing this procedure will be delivered. Its size will depend upon the number of domains to be included in the domain package, as well as each domain's characteristics (type, null-bearing, etc.). The procedure will be written specifically for each application (or set of applications), therefore each procedure is 100-percent new code.

Abstract_Interface_View

Abstract_Interface_View is the second Ada procedure that must be written by the interface programmer. It is specific to the services requested of the data base and particular rows in the data base. Again, it is not a deliverable of this project, but the rules and methods for writing this procedure will be delivered. The procedure will be written specifically for each application (set of applications), therefore each procedure is 100-percent new code.

Generator_Support

Generaator_Support is an Ada package specification that will contain common declarations (generally enumerated types) for the packages to be delivered. It will contain approximately 100 lines of new code. (It is not shown in Figure 2 since it contains only global declarations.)

Abstract_Domain_Generator

Abstract_Domain_Generator is a generic Ada package that contains several layers of nested generics. It will be instantiated by the Domain_View written by the interface programmer. The instantiation will be used to create the domain packages which will be `WITH`ed by the application programmer and the abstract interface. It is a deliverable of this project and will be approximately 325 lines of Ada code. Approximately 15 percent of this code will be reused from the previous STARS contract.

Abstract_Interface_Generator

Abstract_Interface_Generator is a generic Ada package that contains several layers of nested generics. It will be instantiated by the Abstract_Interface View written by the interface programmer. The instantiation will be used to generate the Ada specification and body of the abstract interface module. It is a deliverable of this project and will be approximately 975 lines of Ada code. Approximately 20 percent of this code will be reused from the previous STARS contract.

Domain Packages

The Domain packages will be automatically generated by the instantiation of the Domain View, written by the interface programmer. They will be semantically correct Ada and adhere to the SAME methodology. They will vary in size depending on the number of domains needed by the application.

Base_Specific_Domains

Base_Specific_Domains is an automatically generated Ada package specification that will be used by the Abstract_Interface_Generic. It will vary in size depending on the number of domains needed by the application.

Abstract_Interface Module

Abstract_Interface module is an Ada package (specification and body) automatically generated by the instantiation of the Abstract_Interface generic. It will vary in size depending on the number of SQL services needed by the application and the types of information requested.

SQL Module

SQL Module represents the Ada package specification and its corresponding SQL module, which contains the SQL statements to implement services requested to/from the data base. This package will be written manually for each application (set of applications) and its size will depend on the application.

Application Program

The Ada application program delivered for demonstration will be dependent on the data base services requested. The application will be 100-percent new code.

Ada PACKAGE SPECIFICATIONS DEFINING INTERCOMPONENT INTERFACES

The following listings are examples of the structure outlined in the preceding paragraphs. They are by no means complete but are intended to demonstrate the feasibility of the approach. Specifically, the specifications of the generic packages Abstract_Domain_Generator and Abstract_Interface contain some features that are experimental and omit others that will be needed in production. Specifically, we have not addressed conveying SQL semantic information for generation of the Abstract module bodies. Specific information will be needed to determine the logic of these bodies. Further research and design prototypes will be needed to provide a straightforward implementation for the interface programmer. The bodies of the these packages have not been included, but they will, of course, be deliverable. Since the best way to be precise about the proposed interface is to present compilable Ada specs, the following is given with some explanation.

Domain_View Procedure

The Domain_View procedure is written by the interface programmer to describe the domains to be used by the Ada application. Rules and methods for writing these procedure will be included in the user documentation, CDRL 2020.

```
with generator_support;
use generator_support;
with abstract_domain_generator;

-- This procedure is written by the interface programmer to
-- describe the domains to be used by the application
-- programmer and the abstract interface

procedure domain_view is
  begin
    declare
      type doms is (sno, sname, status, city); -- the domains
                                              --within this domain pkg
-- This generic, abstract_domain_generator, is instantiated once
-- for each domain package

      package domain1 is new abstract_domain_generator
                      ("Suppliers_Definition_Pkg", doms);

      -- This generic, generate_domain is instantiated once for each
      -- abstract domain within the package

      package first is new domain1.generate_domain
            (sno, char, 1, 5, contains_null);
      package second is new domain1.generate_domain
            (sname, char, 1, 20, contains_null);
      package third is new domain1.generate_domain
            (status, int, 0, 100, contains_null);
      package fourth is new domain1.generate_domain
            (city, char, 1,15,contains_null);
    begin
      start_generation; -- procedure call to start generating
            -- the packages
    end;

 -- second domain would go here
 -- third etc ...

end domain_view;
```

Abstract_Domain_Generator

The Abstract_Domain_Generator is a generic package containing levels of nested generics, which when instantiated by the interface program in will automatically generate domain packages needed.

```
with generator_support;
use generator_support;

generic
  domain_package_name: string; -- what the domain package will be
                                -- named
  type domains is (<>); -- one for each domain you will have
package abstract_domain_generator is
  generic
    type_name: domains;
    class: class_types;
    range_start: integer;  -- represents length, range, etc.
    range_stop: integer;
    null_bearing: null_indicator:= contains_null;
  package generate_domain is
  end generate_domain;

  generic
    type_name: nomains;
    class: class_types;
    range_based_on: domains; -- used when this types range is based
            -- on lengths specified by anther domain
    null_bearing: null_indicator:= contains_null;
  package generate_domain2 is
  end generate_domain2;

end abstract_domain_generator;
```

Abstract_Interface_View

The Abstract_Interface_View is written by the interface programmer to describe the records and the procedures to be included in the abstract interface.

```
with base_specific_domains; -- generated automatically
use base_specific_domains;
with abstract_interface_generator;
with generator_support;
use generator_support;

procedure abstract_interface_view is
  type records is (supplier_record_type); -- only one record in
                                          -- this example
  type procs is (acquiresupplier, incrstatus, setstatus);
  -- 3 procedures in the abstract interface specification
  -- The first level of instantiation is for the entire abstract
  -- interface..it is done only once
```

```
package ab_if is new abstract_interface_generator(
    concrete_module_name => "example_c",
    abstract_interface_name => "example_c_module",
    record_names => records,
    procedure_names => procs);
begin
  declare
    type components is (sno, sname, status, city);
    -- components of this record
    -- The following is the instantiation of the record generator
    -- generic..once for each record type declaration in the
    -- abstract interface

    package first_record is new ab_if.record_generator(
      components_in_record => components,
      record_name => supplier_record_type);

    -- The following are instantiations of the component
    -- generator generic..once for each component of this record

    package first_component is new
           first_record.component_generator(
           component_name => sno,
           component_domain_type => sno_not_null);
    package second_component is new
           first_record.component_generator(
           component_name => sname,
           component_domain_type => sname_type);
    package third_component is new
           first_record.component_generator(
           component_name => status,
           component_domain_type => status_type);
    package fourth_component is new
           first_record.component_generator(
           component_name => city,
           component_domain_type => city_type);
    begin
      start_generation; -- calls to initiate the generation of
           -- records
    end;
-- next record here
-- next record here ...

    declare
      type params is (sno, supplier_record, found);

      -- The following instantiation is for procedures within the
      -- abstract interface.  It is done once for each procedure

      package proc1 is new ab_if.procedure_generator(
        procedure_name => acquiresupplier,
        parameters => params);

      -- The following generic is for describing parameters to
      -- this procedure.  Parameter type will determine which
      -- generic is  instantiated.  A generic will be
```

```
        -- instantiated for each parameter.

        package param1 is new
          proc1.params_of_domain_type_generator(params => sno,
          its_type => sno_not_null,
          its_mode => in_param);
        package param2 is new
          proc1.params_of_record_type_generator(params =>
          supplier_record,
          its_type => supplier_record_type,
          its_mode => inout_param);
        package param3 is new
          proc1.params_of_boolean_type_generator(params => found,
          its_mode => out_param);
      begin
      start_generation; -- calls to initiate the generation of
            --procedures
      end;
      -- next procedure here
      -- next procedure here ...

end abstract_interface_view;
```

Abstract_Interface_Generator

The Abstract_Interface_Generator is a deliverable of the contract. It contains levels of nested generics that, when instantiated by the interface program (Abstract_Interface_View), will automatically generate the abstract interface (specification and body).

```
with generator_support;
use generator_support;
with base_specific_domains; -- automatically generated package
use base_specific_domains;

generic
  -- first parameter represents name of the concrete module
  concrete_module_name : string;
  -- second parameter represents name of the abstract i/f
  abstract_interface_name : string;
  type record_names is (<>); -- names of all records to be declared
  type procedure_names is (<>); -- names all procs to be declared

package abstract_interface_generator is

  -- The following generic, record_generator is instantiated once
  -- for each record in the interface

  generic
    -- first parameter represents components in this record
    type components_in_record is (<>);
    record_name : record_names;
  package record_generator is
```

```
    -- This following generic, component_generator is instantiated
    -- once for each component within the record
        generic
          component_name : components_in_record;
          component_domain_type : base_specific_domain_types;
        package component_generator is
        end component_generator;

  end record_generator;

  -- The following generic, procedure generator, is instantiated
  -- once for each procedure in the interface

  generic
       procedure_name : procedure_names;
       type parameters is (<>);
  package procedure_generator is

    -- A generic is instantiated once for each parameter to this
    -- procedure.  The type of the parameter will determine which
    -- generic is instantiated

        generic
          params : parameters;
          its_type : base_specific_domain_types;
          its_mode : mode;
        package params_of_domain_type_generator is
        end params_of_domain_type_generator;

    -- Note this generic is for parameters of type record_type

        generic
          params : parameters;
          its_type : record_names;
          its_mode : mode;
        package params_of_record_type_generator is
        end params_of_record_type_generator;

    -- Note this generic is for parameters of type boolean

        generic
          params : parameters;
          its_mode : mode;
        package params_of_boolean_type_generator is
        end params_of_boolean_type_generator;

  end procedure_generator;

end abstract_interface_generator;
```

Suppliers_Definition_Pkg

The Suppliers_Domain_Pkg is automatically created after execution of the domain_view described earlier. It represents the domain package(s) that the interface programmer and the application programmer will use.

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
with SQL_Int_Pkg; use SQL_Int_Pkg;

package Suppliers_Definiton_Pkg is

  type SNONN_Base is new SQL_Char_Not_Null;
  subtype SNO_Not_Null is SNONN_Base (1..5);
  type SNO_Base is new SQL_Char;
  subtype SNO_Type is SNO_Base (SNO_not_Null'Length);
  package SNO_Ops is new SQL_Char_Ops (SNO_Base, SNONN_Base);

  type SNAMENN_Base is new SQL_Char_Not_Null;
  subtype SNAME_Not_Null is SNAMENN_Base (1..20);
  type SNAME_Base is new SQL_Char;
  subtype SNAME_Type is SNAME_Base (SNAME_not_Null'Length);
  package SNAME_Ops is new SQL_Char_Ops (SNAME_Base, SNAMENN_Base);

  type Status_Not_null is new SQL_Int_Not_null;
  type Status_Type is new SQL_Int;
  package Status_Ops is new SQL_Int_Ops (Status_Type,
                          Status_Not_Null);

  type CITYNN_Base is new SQL_Char_Not_Null;
  subtype CITY_Not_Null is CITYNN_Base (1..15);
  type CITY_Base is new SQL_Char;
  subtype CITY_Type is CITY_Base (CITY_not_Null'Length);
  package CITY_Ops is new SQL_Char_Ops (CITY_Base, CITYNN_Base);

end Suppliers_Definition_Pkg;
```

Abstract_Interface (Ada Spec and Body)

Abstract_Interface (Ada spec and body) is automatically generated from the execution of Abstract_Interface_View described earlier This package is the application programmers interface to the SQL data base. (Note: Only a portion of the generated code is presented here).

```
with suppliers_defintion_pkg;
use suppliers_defintion_pkg;

package example_c_module is
  type supplier_record_type is record
    sno : sno_not_null;
    sname : sname_type;
    scatus : status_type;
    city : city_type;
```

```
  end record;

  procedure acquiresupplier (sno_in : in sno_not_null;
           supplier_record : in out supplier_record_type;
           found : out boolean);

end example_c_module;
```

```
with sql_standard, sql_communications_pkg, sql_database_error_pkg,
example_concrete_module;
use sql_standard, sql_communications_pkg, sql_database_error_pkg,
example_concrete_module;

package example_c_module body is
  use sname_ops, status_ops, city_ops;

  procedure acquiresupplier (sno_in : in sno_not_null;
           supplier_record : in out supplier_record_type;
           found : out boolean) is

    sname_c : char (sname_not_null'range);
    status_c : int;
    city_c : char (city_not_null'range);
    sname_indic, status_indic, city_indic : indicator_type;

  begin
    example_concrete_module.acquiresupplier (char (sno_in),
        char (supplier_record.sno),
        sname_c, sname_indic,
        status_c, status_indic,
        city_c, city_indic,
        sqlcode);
  -- logic of body will be generated here
  end acquiresupplier;

end example_c_module;
```

DEVELOPMENT APPROACH

The CDRL delivery of Ada code for this contract supports iterative prototype development. The successive systems will be developed and delivered in the following manner.

The first iteration of the project will support automatic generation of the Ada specifications for the abstract module and the domain packages. This will be done by writing an Ada procedure that, when executed, will automatically generate the code. The content and semantics of this procedure will be determined during the first iteration also.

The second iteration of the project will support automatic generation of the abstract module body and the SQL concrete module. This will require refining the content and semantics of the procedure developed in the first iteration to contain SQL semantic information specific to an application.

Time permitting, the final iteration of the project will address automatic generation of the the data definition language and an implementation without a module language compiler. Modifying the implementation to support a DBMS that does not include a module language compiler will prove the feasibility of the approach for any SQL-based DBMS, specifically the DBMS chosen for use in the Software Repository.

In general, this contract will explore the feasibility of automatic code generation to support the SAME methodology for accessing an SQL database from within an Ada program. The contract will not attempt to rework or refine the definition of the SAME. There are however, several open issues raised in [1] left to be addressed by specific implementations. These issues are detailed here and our approach to their implementation is given.

Decimal Data Type

ANSI SQL supports the type decimal. The Ada programming language, however supports no directly analogous type. Furthermore, ANSI standard SQL, as described in [2], does not support decimal data in Ada programs. The SAME standard, therefore, does not provide a standard support package for null and non-null bearing decimal types. [1] does detail a method for providing decimal data type support for numeric data coded in binary coded decimal (BCD). This choice was made since any DBMS that supports the decimal type is likely to do so by storing values of the type in the machine's packed or binary coded decimal representation. Support for BCD in SAME is that of an abstract data type whose fundamental operations (arithmetic, comparison, etc.) are provided by assembler-level routines. It is inefficient in comparison to software that might be provided by a compiler that directly supports BCD. Furthermore, this implementation would be specific to each assembly language for each machine. To implement a similar approach for this contract would require learning the assembler language for the VAX workstations hardware and developing routines for decimal support. We believe this approach is too industrious (and completely nonportable) for an 8-month contract. Furthermore, the DBMS used for this contract, VAX RDB, provides limited support for packed decimal type. Because the data bases that underlie VAX SQL do not support the data type, specifying the decimal data type for a column will generate a warning message and create the column with a data type that depends on the precision argument specified:

- decimal(1) through decimal(4) are converted to smallint,
- decimal(5) through decimal(9) are converted to integer,
- decimal(10) through decimal(18) are converted to quadword, etc.

Our implementation will allow this automatic conversion also. Should a standard implementation for decimal support be incorporated into the ANSI standard, its addition to this implementation would be straightforward.

Arbitrary Data Types

SAME provides standard support for types in ANSI standard SQL. Many data base management systems extend the standard to other types. SAME documentation outlines the way a SAME user can extend the data typing facilities. While this approach is not easily automated (and possibly not implementable at all), it is quite straightforward. SAME provides standard support for Ada enumeration types and presents two separate support packages for the date-time data type to demonstrate type extension. Time permitting, a simple record type will be defined within the Ada application domain and support for arbitrary type mapping will be developed.

Using SAME Without a Module Language Compiler

The SAME approach assumes the existence of an SQL module language compiler. However, the approach may still be used in environments where no such compiler exists. In fact, the STARS repository data base and associated environment may not include an SQL module language compiler. Therefore, this project will simulate the use of a module language compiler for some data base as deemed appropriate by IBM. Earlier discussions have suggested Oracle as such a possibility. Therefore, initial research will head in that direction. As an interesting note, when researching Ada interfaces available with Oracle, such as Pro*Ada, we found that the decimal data type, as discussed previously, is not supported by the Oracle Call Interface, further substantiating our decision not include specific support for this type in the Ada program.

REFERENCES

[1] M. H. Graham. 1989. *Guidelines for the Use of the SAME.* SEI Technical Report CMU/SEI-89-TR-16 and ESD-TR-89-24. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

[2] *Database Language—SQL.* American National Standards Institute, 1986. X3.135-1986.